Yoav Weiss - Mar 16, 2023 Chromium.org

AsyncContext is a TC39 proposal aimed at improving the debuggability and traceability of async tasks.
As such, it has some overlap with TaskAttribution, but in a way that's exposed to JS developers and to the web.
Thinking about how these two technologies interact and how AsyncContext can be layered on top of TaskAttribution, it seemed to me that it may be easiest to think of AsyncContext not as a JS language feature, but as a web platform feature. This (extremely short) document sketches out how that may look like, and how one may be able to implement that in Chromium.

Web API

IDL

```
[Exposed=(Window, Worker)]
interface AsyncContext {
   constructor();
   void run(any value, callback func);
   any get();
   static void wrap(callback func);
}
```

Implementation on top of TaskAttribution

Below is an *extremely* high level sketch, which may even work:

- AsyncContext would be a platform object which keeps a map of TaskAttributionID to value objects.
- `run()` call would start a new TaskScope (or a similar class, as it won't really be starting a new task) and add the value object to the above map in the current task ID.
- `get()` would go up the ancestor chain, check the map for value objects that fit the task ID and return the first one it gets.
- `wrap()` would set the current task ID on the callback as its parent.

Gotchas

I *think* the above should work with nesting, but it should be verified.

Downsides

It’s unclear how non-browser environments would implement the same. Would they need to implement TaskAttribution as part of the host?

JS language feature

We can also imagine what happens if AsyncContext is a JS language feature, and how TaskAttribution could have been built on top of it, at least theoretically.

In such a scenario, we wouldn’t use any of the existing TaskAttribution machinery for the implementation, but instead have V8 propagate objects on every microtask (potentially multiple objects?), instead of the current ID integer.
`wrap()` calls would hang those objects on callbacks as well.

Then for every web API that handles an async task, we’d need to wrap the passed callback before passing it in V8 bindings. For blink-side generated promises, their context would need to be determined and set at creation time.

TaskAttribution could then rely on that concept, and set an internal AsyncContext at points that it may want to keep track of further down the line.

Downsides

- TaskAttribution would no longer be able to keep track of all tasks, but would need to know ahead of time which tasks it wants to track. That may be fine for current use cases, but we need to examine future ones in that light.
- V8 would potentially need to pass around a lot more data with each Promise and bound callback. That cost may be incurred only when AsyncContext is used (?)